PorePy: Simulation software for mixed-dimensional problems Part II: Under the hood

Eirik Keilegavlen

INSPIRE Summer School – Geilo August 2019

Goal of part II

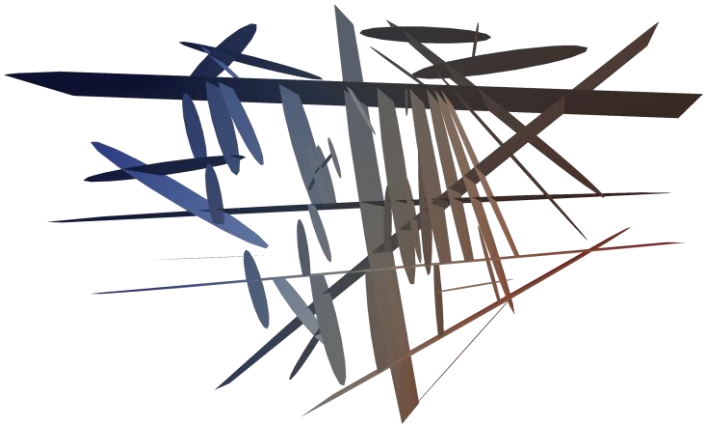
1. Give an understanding of main conceptual challenges in mixed-dimensional simulations, independent of modeling and simulation framework.
2. Show modeling and design choices that underpin PorePy, and are transferrable to other simulation frameworks
3. Illustrate more advanced use of PorePy:
    i. Mesh construction, including non-matching grids
    ii. Multi-physics problems

Outline Part II:

- Geometry
- Meshing
- Grid data structure
- Coupling between subdomains
- Parameter assignment
- Local discretization
- Global discretization

Geometry and meshing

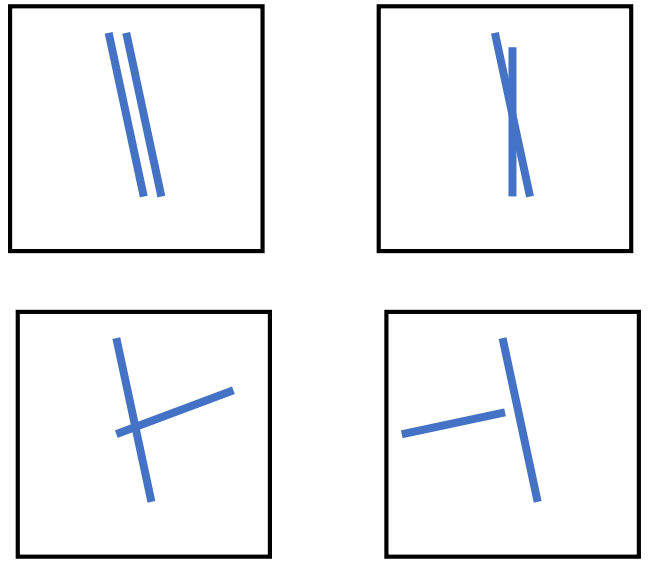
Mesh construction



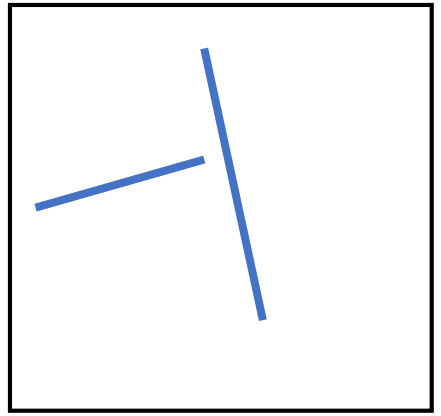
Fracture network geometries can be arbitrarily ill-suited for meshing.

Two main reasons not to use mixed-dimensional approach:

- The mechanics of mesh construction is highly technical
  - High quality meshing software is available (Gmsh), but requires specific input
  - The computational geometry tasks before and after meshing are non-trivial
- The high cost of computations on a geometry-resolving mesh

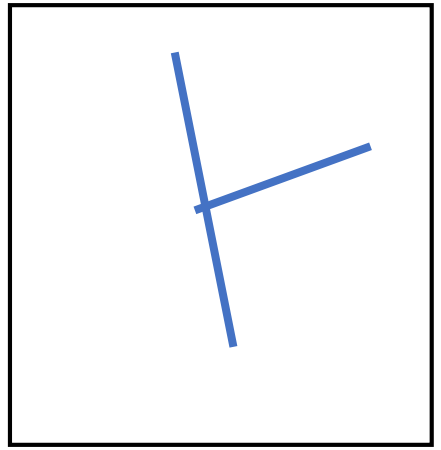


Resolutions and tolerances



At least three levels of resolution during geometry processing

- Accuracy of geometric computations ($\epsilon_{machine}$)
- Accuracy of the data (should two almost intersecting lines be merged?)
- (Local) target mesh resolution

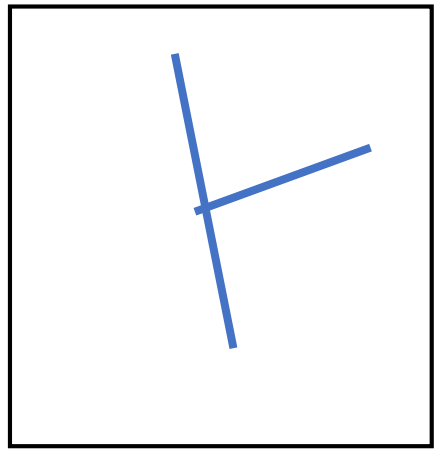


Upshot: What is a geometric point?

Decisions must be made automatically.

PorePy treatment of fractures

- Meshing of complex domains is fairly stable (both 2d and 3d)
- Approach: Strictly adhere to the specified fracture geometry
  - Fine details will be resolved, at the cost of many or badly shaped cells.
  - Some tools are available for snapping geometry etc.
- Difficult problems are difficult



Specification of fracture geometry

- PorePy treats fractures, collected into fracture networks
- Specification of fractures is different in 2d and 3d, the underlying implementation is completely different
- Main classes for fracture network manipulation including meshing are FractureNetwork2d and FractureNetwork3d (next slide)
- Import filters from csv files to networks are available (`pp.importer`)
- Fractures should be planar, convex objects.

Define a fracture network

2d

```
# fractures are specified by points and
# connections
pt = np.array([[0.2, 0.2], [0.7, 0.8]]).T
con = np.array([[0], [1]])
domain = {‘xmin’: 0, ‘xmax’: 1,
          ‘ymin’: 0, ‘ymax’: 1}

network = pp.FractureNetwork2d(pt, con, domain)
# generation of mixed-dimensional mesh
mesh = network.mesh(…)
```

3d

```
# Specific class for 3d fractures
# Specify a fracture by its vertexes
fracture = pp.Fracture(np.array([[0, 1, 1, 0],
                [0, 0, 1, 1], [0, 0, 1, 1]]))

domain = {‘xmin’: 0, ‘xmax’: 2, ‘ymin’: 0,
          ‘ymax’: 3, ‘zmin’: 0, ‘zmax’: 2}

network = pp.FractureNetwork3d(fracture, domain)
mesh = network.mesh(…)
```

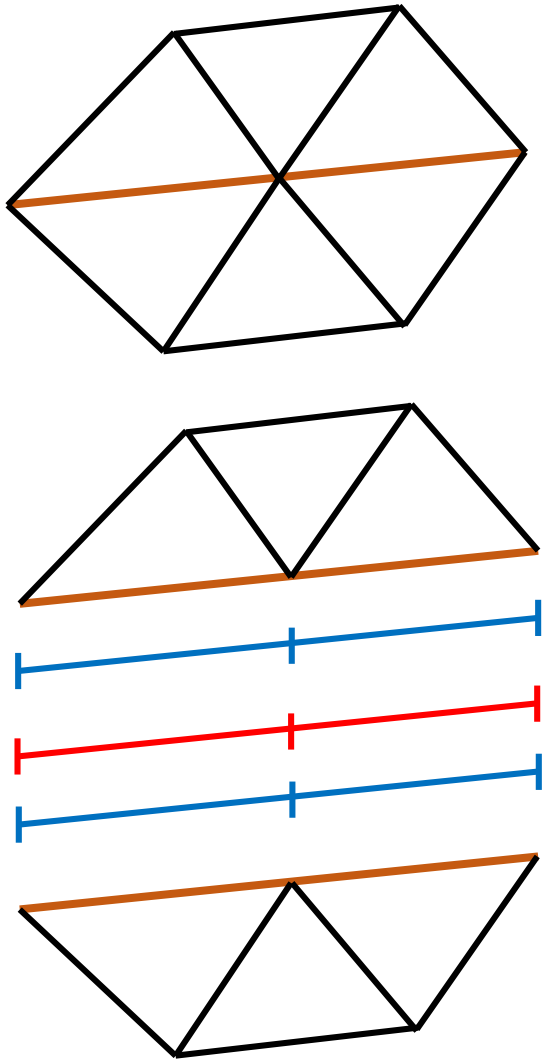
Mesh size control

- Gmsh associate a user provided a target mesh size to all points in the geometry description.
  - Interpreted as guidance / wish, not a rule
- PorePy attempts to compute this for a specific geometry, based on three parameters:
  - The desired mesh size on fractures (one value for all fractures)
  - The desired mesh size on the boundary (representing far-field conditions)
  - The minimal target mesh size to be sent to gmsh
- Depending on the fracture geometry and the specified value, the mesh produced by gmsh may or may not adhere to these parameters

Grids of different dimensions

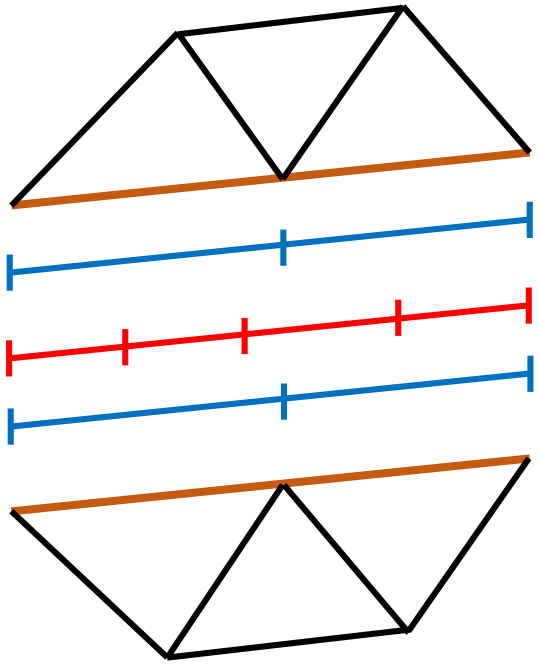
From 3d mesh to mixed-dimensional mesh

- Gmsh returns one 3d mesh that conforms to all fractures, intersections etc.
- Conversion into mixed-dimensional mesh requires
  - splitting of nodes and faces,
  - introduction of lower-dimensional grids
  - identification of mappings between the grids
  - introduction of a mortar grid on interfaces
- PorePy takes care of this behind the sense
  - Robust but somewhat time consuming



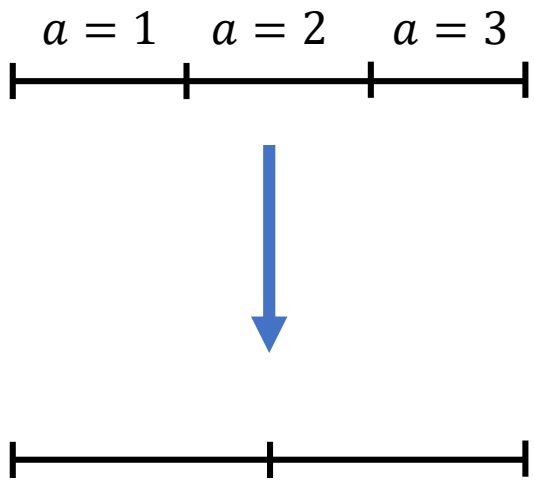
Non-matching grids

- PorePy handles non-matching grids between interfaces and subdomains
- Gmsh generates matching grids, meshes on subdomains and interfaces can be updated
- Mappings between grids are automatically updated
- NB: Non-matching grids increase the chance of unstable discretizations (inf-sup)



Projections

- Variables are projected between subdomain grids and interface (mortar) grids
- Different projection operators for extensive (e.g. flux) and intensive (e.g. pressure) quantities
- Projection objects reside on the mortar grid



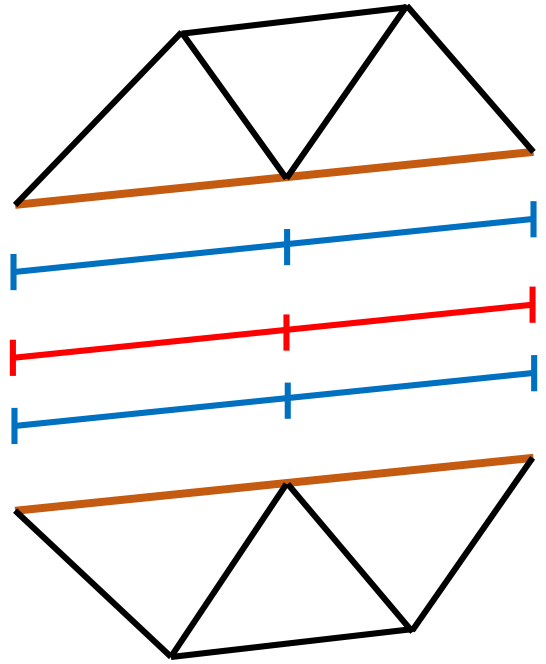
Intensive	$\Pi a = 4/3$	$\Pi a = 8/3$
Extensive	$\Pi a = 2$	$\Pi a = 4$

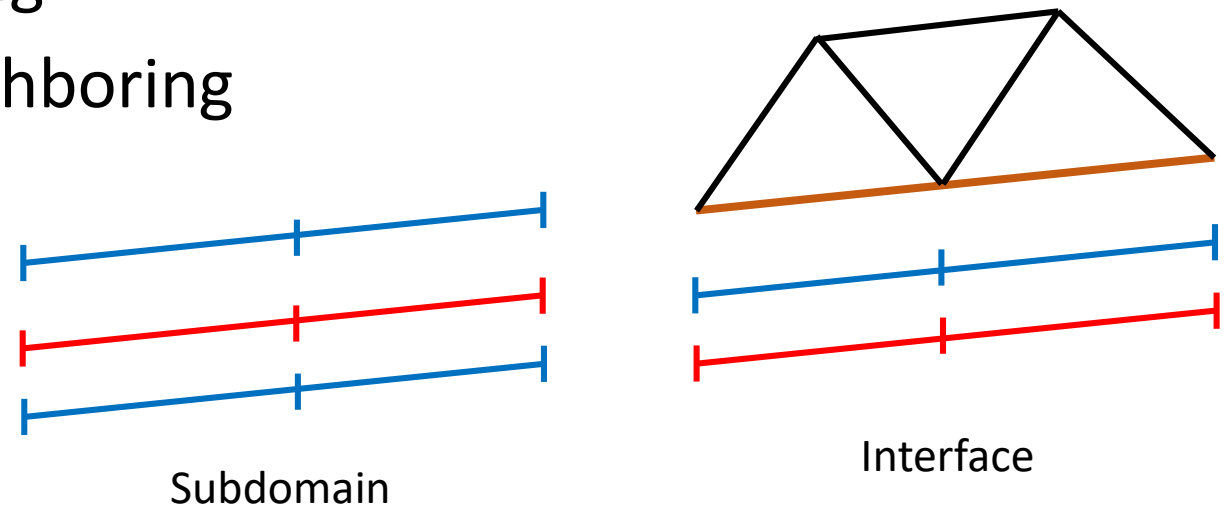
Permissible dynamical couplings

Coupling between subdomains

Modeling choice: Conditions on the coupling between subdomains:

1. Only coupling between subdomains one dimension apart
2. Coupling between subdomains only via the shared interface – no direct coupling
3. The interface only sees its two neighboring subdomains





Comments

- 3d-1d couplings (e.g. roots) are not permitted
- 3d-3d couplings not accounted for, but can probably be hacked
- Couplings via interface variables
  - Very often, this is a flux of a conserved quantity
- Structure of linear system is rather sparse
- Reuse of discretization

Darcy's law:

$$\boldsymbol{u}_i + K_i \nabla p_i = \mathbf{0}$$

Conservation of mass:

$$\nabla \cdot \boldsymbol{u}_i = f_l + \sum_{j \in \widehat{S}_i} \Xi_j^i \lambda_j$$

Boundary condition:

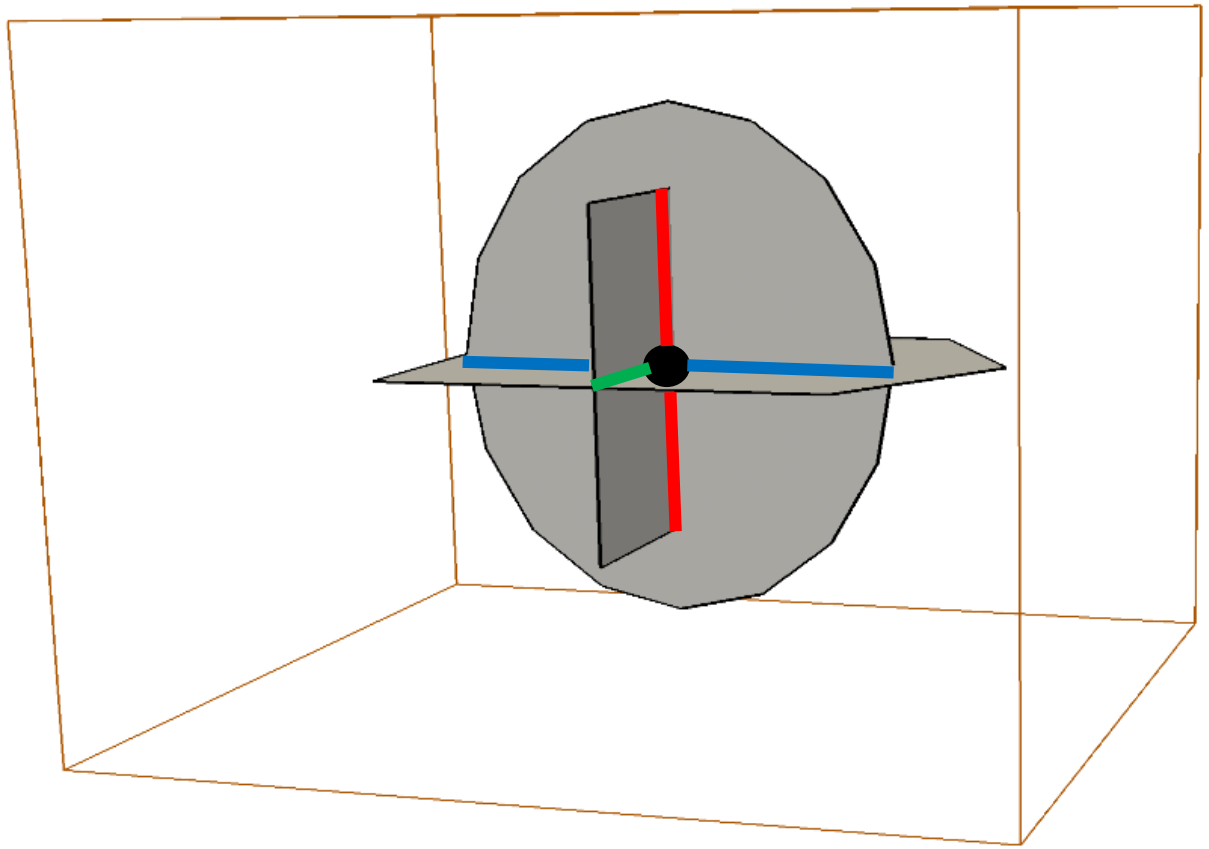
$$\boldsymbol{u}_i \cdot \boldsymbol{n}_j = \Xi_j^i \lambda_j, \qquad j \in \check{S}_j$$

Interface:

$$\lambda_j + \kappa_j \left(\Pi_l^j p_l - \Pi_h^j tr\, p_h \right) = 0$$

Mixed-dimensional data structure

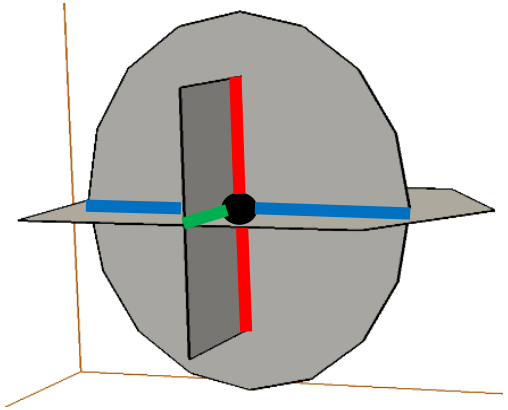
Mixed-dimensional grid data structure

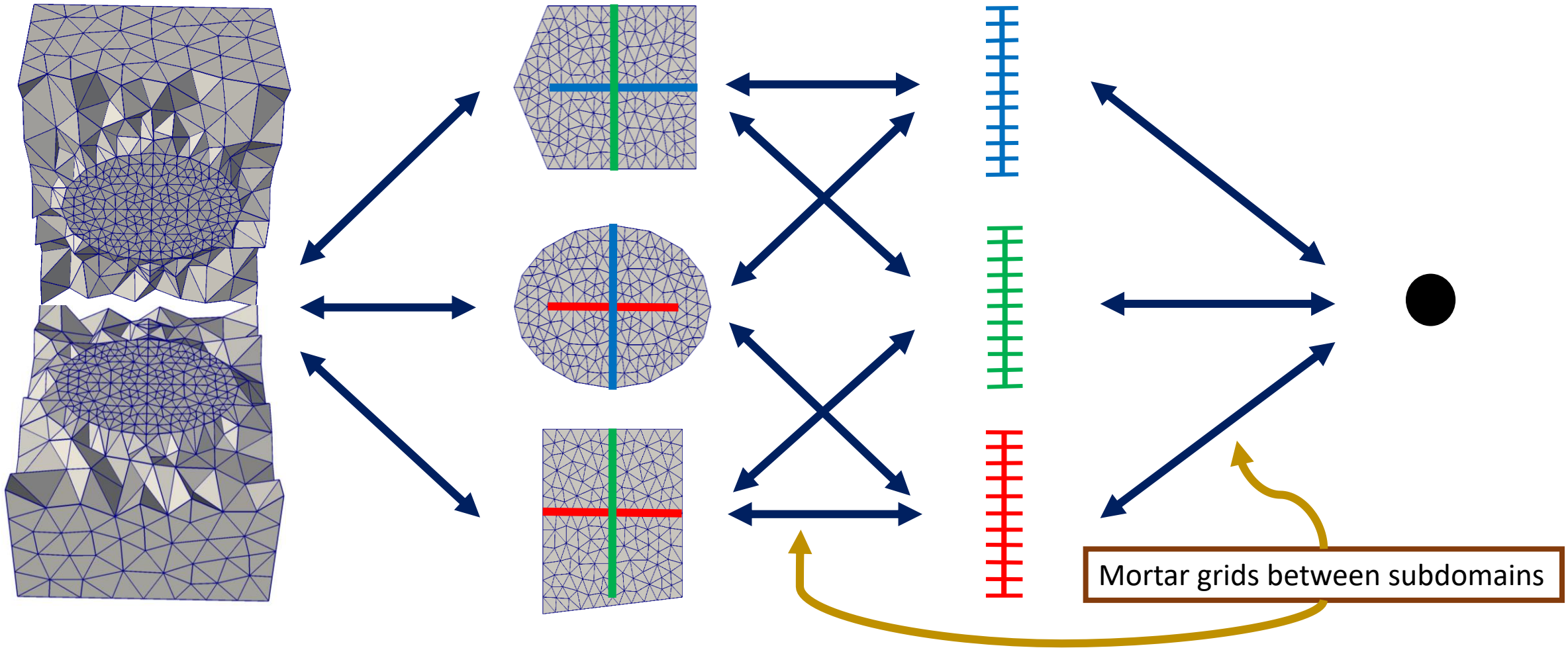


3 intersecting planes embedded in 3d domain:

- 3 2d objects
- 3 1d intersection lines (colored)
- 1 0d intersection of intersections

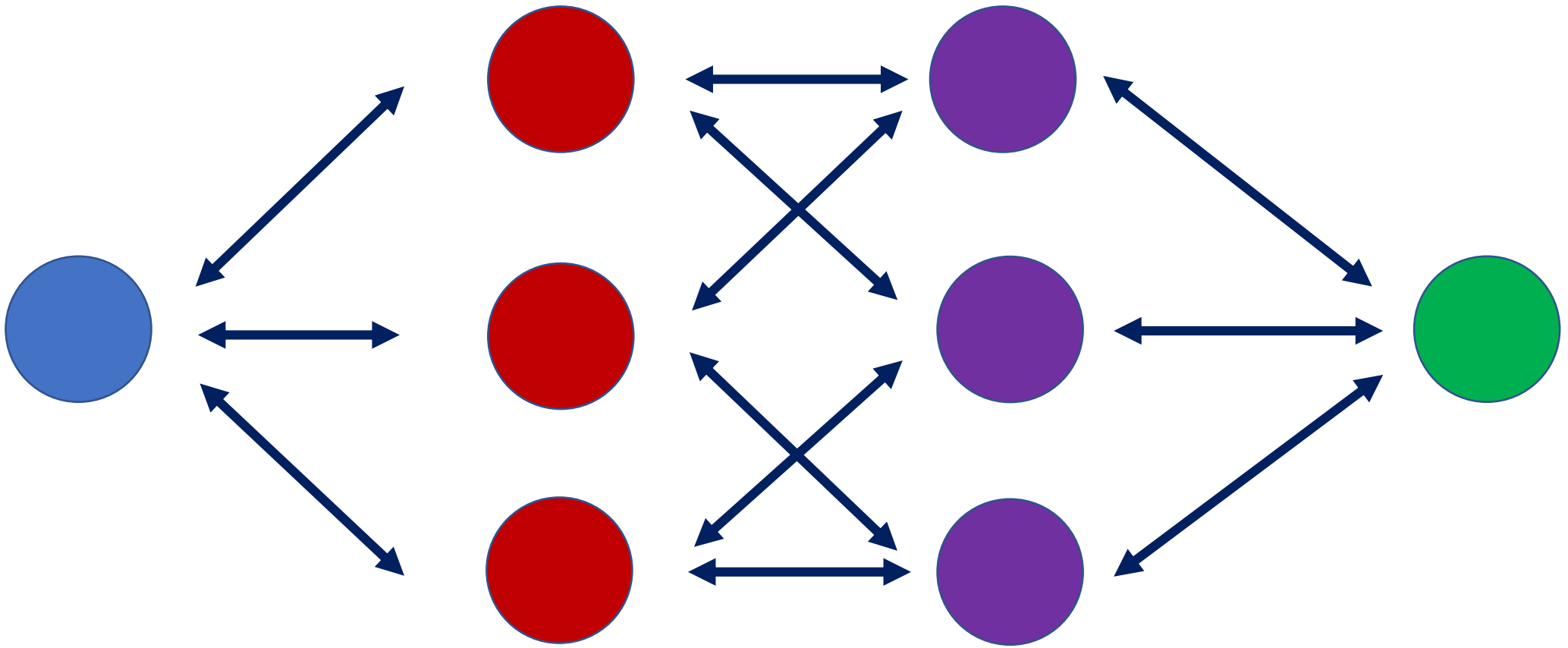
Co-dimension 1 couplings





Graph representation

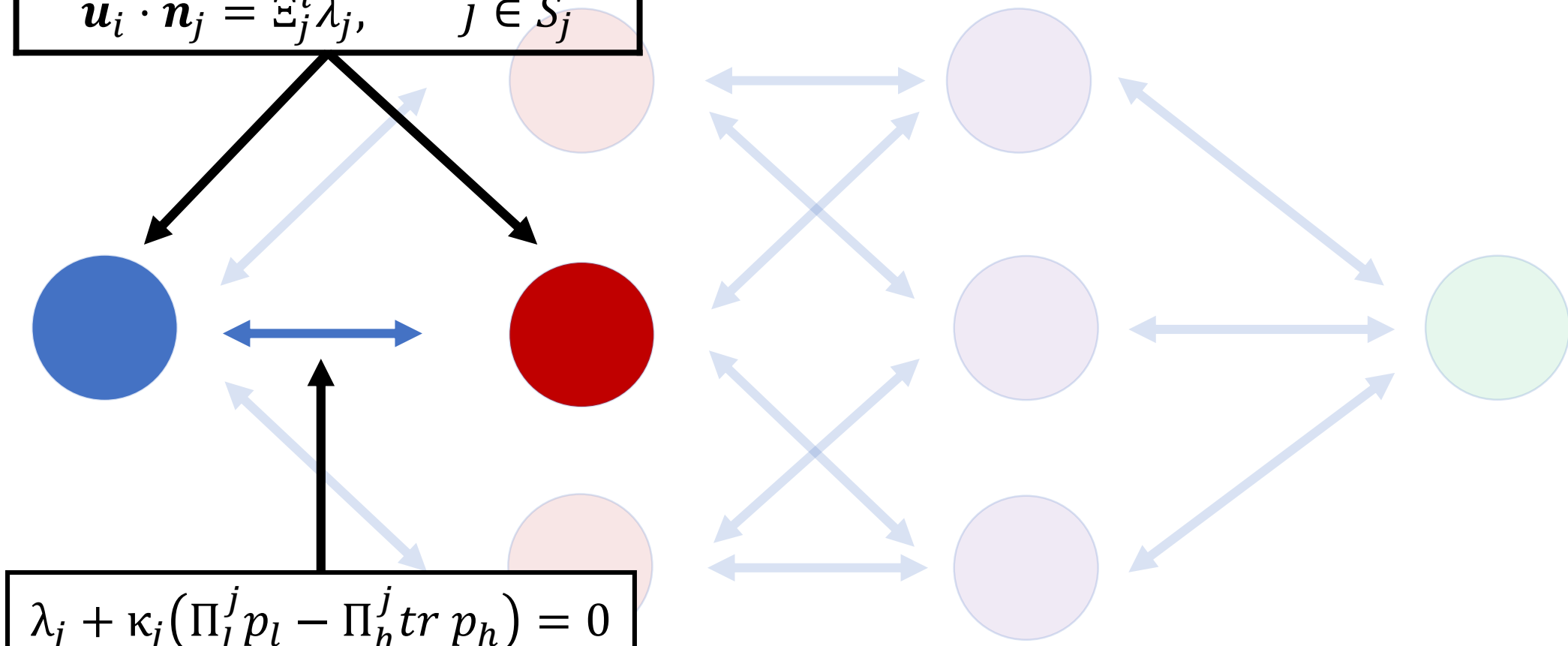
Each node represent a simulation domain
Coupling condition on edges



$$\boldsymbol{u}_i + K_i \nabla p_i = \mathbf{0}$$

$$\nabla \cdot \boldsymbol{u}_i = f_l + \sum_{j \in \widehat{S}_i} \Xi_j^i \lambda_j$$

$$\boldsymbol{u}_i \cdot \boldsymbol{n}_j = \Xi_j^i \lambda_j, \qquad j \in \check{S}_j$$



$$\lambda_j + \kappa_j \left(\Pi_l^j p_l - \Pi_h^j tr\, p_h \right) = 0$$

Equations and discretization

Discrete equations

- Variables and discrete equations reside on subdomains and interfaces (graph nodes and edges)
- PorePy offers complete flexibility (sometimes to an annoying degree) in defining:
  - variables on subdomains and interfaces
  - couplings between and within graph components (can be a bit cumbersome)
- Discretizations are defined in a per-term-per-variable manner
- Example: Time-dependent advection diffusion problems are composed of three subdomain discretizations: Accumulation, advection, diffusion.

Identifiers for variables, discretizations and parameters

- Variables:
  - should have a name,
  - Will have a number of dofs per subdomain / interface
- Discretizations:
  - operate on single terms in an equation
  - are associated with variables or coupling of variables
  - need parameters
- Parameters:
  - are specific to the subdomain grid, and term in the equation

Identical setup on all subdomains should be simple, while flexibility should be preserved.

```
# Parameter definition
param_key = ‘flow_parameters’  # Parameters are identified by a string
for g, d in gb:
    # populate d[pp.PARAMETERS][param_key]
    pp.initialize_data(g, d, param_key, …)
    # pp.initialize_default_ data(g, d, ‘flow’, .., param_key)

# Define variable and discretization
variable = ‘pressure’  # a variable is identified by a string
discr = pp.Mpfa(param_key)  # discretization object knows its parameter key

for g, d in gb:
    # Define a variable on the grid, and its number of dofs
    d[pp.VARIABLES] = {variable: {‘cells’: 1, ‘faces’: 0}}  # define variable
    # Associate a discretization with each term associated with the variable
    d[pp.DISCRETIZATION] = {variable: {‘diffusion’:  discr,}} # ‘advection’: pp.Upwind()
```

Discretizations

- Discretizations operate on individual subdomains or edges
- Subdomain discretization classes are available for the most common (to the developers) operations:
  - Scalar elliptic equations: Finite volumes, mixed finite elements, mixed virtual elements
  - Advection equations: Single point upwind methods
  - Elasticity and poro-elasticity: Finite volumes
  - Fracture deformation: Contact mechanics by variational inequalities
- Implementation is (almost) that of a fixed-dimensional problem
- Interface discretization: Mainly P0

Assembly

- An assembler object is responsible for global discretization and system assembly
- The assembler defines variable numbering
  - block_dof: One per variable per subdomain / interface
  - full_dof: Corresponding to rows

```
# define parameters, variables, discretizations
…

#
assembler = pp.Assembler(gb)
# Generate discretization matrices
assembler.discretize()
# Assemble linear system
A, b = assembler.assemble_matrix_rhs()

x = spsolve(A, b)

# Distribute the variables on the graph
assembler.distribute_variable(x)

# visualization?
```

Visualization

- Export filter to Paraview is available: `pp.Exporter(gb)`
- Grids of all dimensions are stored in a .pvd container
- Filtering by grid dimension in Paraview

(Intentionally?) missing pieces

- Modules for setup and discretization of standard equations
  - Cut-paste-modify is currently overused.
- Solvers:
  - Main linear solver is direct – experiments with more advanced solvers are promising
  - Very limited support for, and experience with, non-linear problems